TUNED AND ASYNCHRONOUS STENCIL KERNELS FOR CPU/GPU SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Sundaresan Venkatasubramanian

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
Computer Science

Georgia Institute of Technology
May 2009

TUNED AND ASYNCHRONOUS STENCIL KERNELS FOR CPU/GPU SYSTEMS

Approved by:

Prof. Richard Vuduc, Advisor
College of Computing
Georgia Institute of Technology

Prof. Hyesoon Kim
College of Computing
Georgia Institute of Technology

Prof. Jeffrey Vetter
College of Computing
Georgia Institute of Technology

Date Approved: May, 05, 2009

To my parents, sister, close friends and the Paramacharya of Kanchi

ACKNOWLEDGEMENTS

I thank my advisor Prof. Richard Vuduc for his invaluable guidance in the making of this thesis. His constant motivation, encouragement, and valuable feedback have greatly helped me in the completion of this thesis. I am deeply grateful to him for providing the great opportunity to work under him.

I would like to thank Prof. Hyesoon Kim and Prof. Jeffrey Vetter for agreeing to be in my committee, for their support and enthusiasm they showed in my work.

I am very grateful to my grandparents for their love and blessings and a special thanks to my sister for her love and support. I wish to thank Karthi anna and Maadhu anna for their support and inspirations right from my school days. Lastly, and most importantly, I owe my parents, S. Venkatasubramanian and T. P. Kalyani much of what I have become. I dedicate this work to them, to honour their love, patience, and caring during these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

We describe heterogeneous multi-CPU and multi-GPU implementations of Jacobi's iterative method for the 2-D Poisson equation on a structured grid, in both single- and double-precision. Properly tuned, our best implementation achieves 98% of the empirical streaming GPU bandwidth (66% of peak) on a NVIDIA C1060. Motivated to find a still faster implementation, we further consider "wildly asynchronous" implementations that can reduce or even eliminate the synchronization bottleneck between iterations. In these versions, which are based on the principle of a chaotic relaxation (Chazan and Miranker, 1969), we simply remove or delay synchronization between iterations, thereby potentially trading off more flops (via more iterations to converge) for a higher degree of asynchronous parallelism. Our relaxed-synchronization implementations on a GPU can be 1.2–2.5× faster than our best synchronized GPU implementation while achieving the same accuracy. Looking forward, this result suggests research on similarly "fast-and-loose" algorithms in the coming era of increasingly massive concurrency and relatively high synchronization or communication costs.

CHAPTER I

INTRODUCTION

1.1 Introduction and Scope

This study began with what we thought would be a trivial exercise: given a bandwidth-rich GPU platform, take a memory-bound computation with a regular memory access pattern and produce a code that runs at the memory bandwidth limit. In particular, we considered the problem of Figure 1.1, a textbook instance of Jacobi's method for a centered finite-difference approximation of the 2-D Poisson equation on a square domain, regularly discretized [9, Chap. 6]. At first glance, the code and data access would appear trivial to implement on a GPU.

Contrary to our expectation, we found that achieving a very high-level of performance in practice—*i.e.*, running near the bandwidth limit using the high-level CUDA programming model [1]—requires a carefully designed implementation. The first contribution of this work is to describe some of the tuning lessons we learned along the way. A second related contribution is our extension of our initial GPU-only implementation to the hetergeneous multi-GPU and hybrid multi-CPU/multi-GPU cases. These extensions are accompanied by predictive performance models that help decide when a hybrid implementation will pay off.

Though we believe these implementations perform well, they are still limited by a fundamental bottleneck: the cost of synchronization. Looking forward, we expect this cost will only get worse as core counts and core heterogeneity increase. These observations motivate the third contribution of this work, which is to "throw out" the deterministic synchronization and replace it with non-deterministic asynchronous parallelism. This technique exploits our specific computation, but yields 1.2–2.5×

Problem: Solve Poisson's equation in 2-D on a square grid:

$$\begin{aligned} -\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)u(x,y) &= f(x,y), \\ 0 < x,y &< 1, \\ u(0,y)=u(x,0) &= 0 \end{aligned}$$

Centered finite-difference approximation on a $(N+2)\times(N+2)$ regular grid with step size h:

$$4\cdot u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 \cdot f_{i,j} + O(h^2) \tag{1}$$

Jacobi's method for this problem and approximation:

$U^0 \leftarrow 0$ // **Initializes an** $(N+2)\times(N+2)$ **grid**
// **For each iteration,** t **(**T **iterations in all)**
for $t \leftarrow 1, 2, \ldots T$ **do**
  for $i \leftarrow 1 \ldots N$ **do**
    for $j \leftarrow 1 \ldots N$ **do**
      $U^{t+1}_{i,j} \leftarrow \frac{1}{4}\cdot(U^t_{i+1,j} + U^t_{i-1,j} + U^t_{i,j+1} + U^t_{i,j-1} + h^2 \cdot f_{i,j})$
    end for
  end for
end for

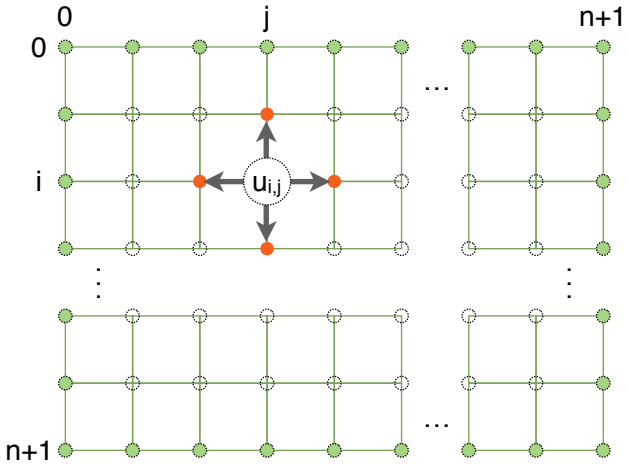


Figure 1: Jacobi's method for the 2-D Poisson equation. (*Top*) Continuous problem, discrete approximation, and solver pseudocode. (*Bottom*) Illustration of the discrete grid and nearest-neighbor dependences.

speedups while achieving the same level of accuracy.

This specific idea is not new; Chazan and Miranker suggested it in their seminal 1969 paper on *chaotic relaxation* [7]. Most recently, researchers have revisited chaotic relaxation in the context of heterogeneous clusters and grid environments [10]. If a GPU reflects possible future multi- and many-core architectural designs, our positive results on GPUs suggest that the pursuit of similarly "fast-and-loose" algorithms will be a fruitful direction.

The scope of this work is limited in at least three significant ways. First, we consider a relatively simple kernel. Whether the tuning techniques and more radical unsynchronized implementations could apply more broadly is unknown. Secondly, our experimental results for the hybrid CPU/GPU implementation do not show significant speedups. Nevertheless, we believe these results are due to our specific evaluation hardware; our performance models suggest pay-offs should be possible on other configurations. Finally, the unsychronized methods have inherent non-determinism. They depart from the baseline parallel implementation in ways that require careful mathematical justification and analysis, discussed in the related work (Section 1.2). This focus of this work is on *potential performance gains*, to suggest the utility of considering loose synchronization in other problem and architectural contexts. Many of the results in this document have appeared in a recent publication [17].

1.2 Related work

A number of other researchers are investigating stencil kernels on GPUs. In the most extensive recent study of which we are aware, Datta, *et al.*, tune 3-D stencil kernels for a broad variety of multicore platforms, including GPUs [8]. In an unpublished manuscript, Giles reports a high fraction (75%) of sustained bandwidth for a 3-D kernel as well on a GPU platform [11]. Amorim, *et al.*, consider the 2-D case as we do here, though they focus on the 9-point stencil and consider a subset of the tuning space

that we consider [2]. Looking beyond regular stencil kernels, some researchers have considered more irregular general sparse matrix kernels and solvers for GPUs [4]. Our paper differs from these primarily in that (a) we consider algorithmic variations via loosely synchronized designs; and (b) we develop hybrid CPU/GPU implementations, in addition to GPU-only versions.

The concept of chaotic relaxation, also called *asynchronous iteration*, has a long history but is largely considered "outside" mainstream numerical methods because of the use of non-determinism [7, 3, 10, 16]. Researchers have considered these methods for use in heterogeneous parallel systems, including grid systems, as noted in the survey paper by Frommer and Szyld (2000) [10]. We show these basic techniques are relevant to GPUs, further suggesting that the techniques will become only more relevant to future many-core systems.

Though our code is non-deterministic—but ultimately still converging-, other researchers have strongly argued that asynchronous algorithms are necessary for current and future multicore platforms even in the deterministic case [6, 5]. The philosophy and results of our paper are aligned with these views.

Looking to alternative paradigms, a related emerging body of work attempts to formulate numerical solvers for time-dependent partial differential equations based on *parallel discrete-event simulation*, including methods known as *asynchronous variational integrators* [15, 14, 13, 12]. These formulations result in similarly asynchronous parallel behavior, and so we regard these methods as another promising class of approaches for future multi- and many-core systems.

CHAPTER II

AN OVERVIEW OF CUDA

This chapter gives a quick overview about CUDA. Readers who are comfortable with CUDA programming may skip this chapter. For a more detailed discussion refer the NVIDIA CUDA programming guide [1].

2.1 CUDA programming model

CUDA is a parallel programming model and software environment that extends C language for writing code on NVIDIA GPU's. A function called "kernel" is executed parallely by multiple threads on the Nvidia GPU. The CPU is generally referred as *host* and has the GPU as an attached coprocessor reffered as the *device*. The C program executes on the host and the kernel code executes on the GPU. Typically the data has to be transferred to the *device* from the *host*, the computation is done on the textitdevice and the result is transferred back to the *host*.

The threads are grouped into group of threads called *thread blocks* which can be one-dimensional, two-dimensional or three-dimensional. A shared memory is available for the threads within a block to cooperate among themselves by sharing data and synchronizing their execution to coordinate memory accesses. The shared memory is a low-latency memory for each processor core and can be compared to an L1 cache. The synchronization mechanism at thread-block level is lightweight, and all threads of a block are expected to reside on the same processor core. A kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. The thread blocks in turn are arranged as one-dimensional or two-dimensional grid of thread blocks as shown in Figure 2.

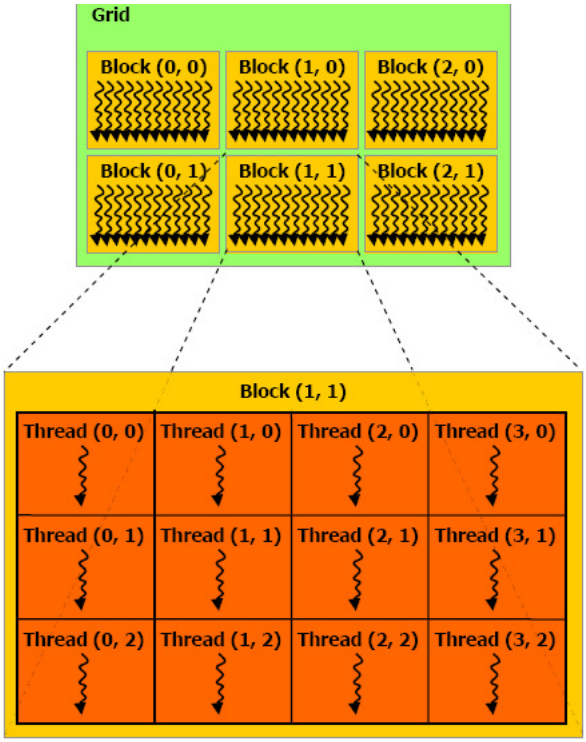


Figure 2: Grid of thread blocks

There is no assurance regarding the order of execution of thread blocks. So the correctness of an algorithm should not depend on the order of execution of the thread blocks and it must be possible to execute them in any order, in parallel or in series. This helps programmers in writing scalable code and the thread blocks to be scheduled in any order across any number of cores. The number of thread blocks in a grid does not depend on the number of processors but rather depends on the size of the data. In many cases the number of blocks is much higher than the total number of processors.

CUDA has a heirarchical memory organization as illustrated by Figure 3. Individual threads have a private local memory and each thread block has a shared memory accessible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory. There is also a read-only constant memory space and a read-only texture memory space which are accessible by all threads.

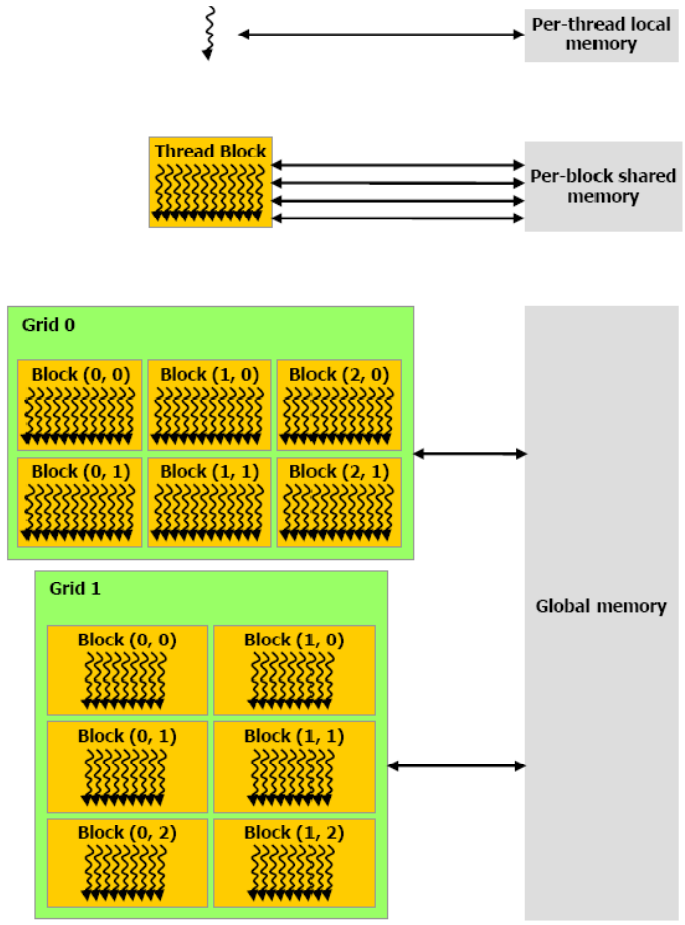


Figure 3: Memory Hierarchy

To manage hundreds of threads running several different programs, the multiprocessor employs an architecture called SIMT (single-instruction, multiple-thread). The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state [1]. The multiprocessor manages threads in groups of 32 parallel threads called warps. A half-warp is either the first or second half of a warp. A multiprocessor splits the given thread block(s) into warps that get scheduled by the SIMT unit. Each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

The number of blocks a multiprocessor can process simultaneously depends on the number of registers per thread and the amount of shared memory per block required for a given kernel since the multiprocessor's registers and shared memory are split among all the threads of the batch of blocks. The kernel will fail to launch if there are not enough registers or shared memory available per multiprocessor to process at least one block. A multiprocessor can execute as many as eight thread blocks concurrently.

2.2 Performance guidelines

In general, the following are some performance guidelines/issues to keep in mind while writing CUDA code:

1. Occupancy - Each hardware multiprocessor has the ability to actively process multiple blocks at one time. The blocks that are processed by one multiprocessor at one time are referred to as active. The number of active blocks is limited by the number of registers per thread and the amount of shared memory per block required by a given kernel. Kernels with minimal resource requirements can better utilize (or occupy) each multiprocessor because the registers and shared memory of the multiprocessor are split among all the threads of the active blocks. The CUDA occupancy calculator can be used to find trade-offs

between number of threads and active blocks versus the number of registers and amount of shared memory. Using the right combination can greatly increase the performance of your kernels.

2. Exploit Shared memory/Registers usage - Shared memory and Registers are much faster than global memory and they should be utilized to the maximum extent wherever possible.

3. Coalescing global memory accesses - Global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction. The requirements for coalescing differs depending on the compute capability of the device.

4. Bank Conflicts - To achieve high memory bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests.

5. Branch statements - Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a datadependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute

independently regardless of whether they are executing common or disjointed code paths.

6. Minimize Host-Device transfers - Host to Device transfer bandwidth is generally very less compared to the device to device transfer bandwidth and it is good avoid transferring data between the host and device unless required.

CUDA profiler (available in Nvidia's site) is a useful tool and can be used to profile different parameters for the above performance issues.

CHAPTER III

BASELINE CPU AND SINGLE-GPU IMPLEMENTATIONS

In this chapter, we describe different tuning techniques we used for our baseline CPU and single-GPU implementations.

3.1 CPU baselines

We consider both sequential and parallel Pthreads-based implementations as our CPU-based baselines. We use a 2-D block partitioning of the domain in the parallel case. The single-program multiple data (SPMD) style pseudocode for each thread of the parallel implementation, which executes T Jacobi iterations, is as follows:

1: **for** $t \leftarrow 1, 2, \ldots, T$ **do**
2: Update my block, b: $U_b^{\text{new}} \leftarrow$ **update**(U_b^{cur})
3: **barrier()**
4: Logically swap U_b^{new} and U_b^{cur} (*i.e.*, swap pointers)
5: **end for**

The explicit barrier in line 3 ensures the parallel code computes the same result (to within round-off) as the sequential code in Figure 1.1.

For the CPU implementation, we perform a "minimal" tuning as follows.

First, we write the code so that the compiler SIMD vectorizes it. We verify this vectorization by inspecting the assembly code.

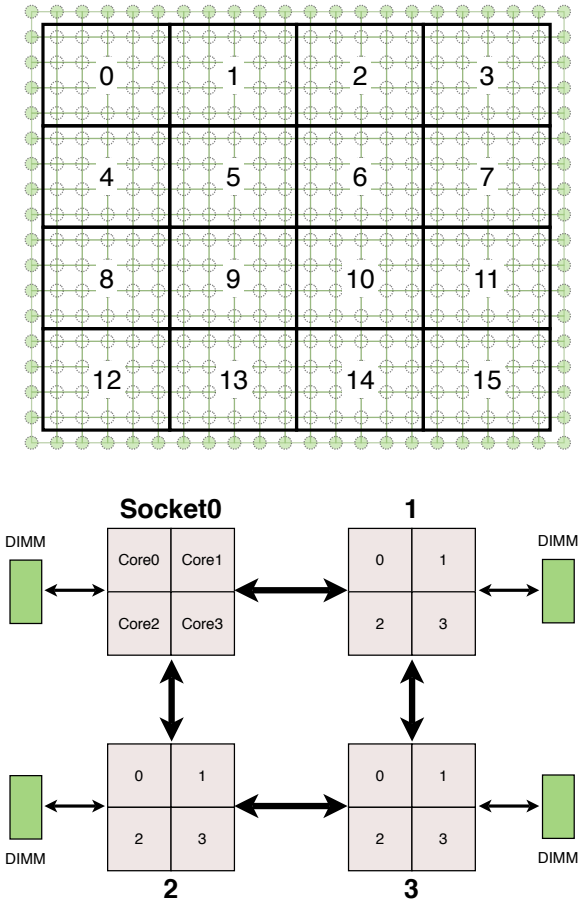


Figure 4: (*Left*) A 16×20 grid of unknowns (plus boundaries) partitioned into a 2-D blocked grid of 4×5 unknowns per block. (*Right*) A quad-socket NUMA system with quad-core processors.

Secondly, we bind threads to cores, *e.g.*, using the Linux affinity scheduling routines. For our experimental evaluation, we consider both explicitly bound and unbound cases.

Thirdly, for the non-uniform memory access (NUMA) multicore CPU architecture used in our evaluations, we allocate the per-thread data blocks, U^{new} and U^{cur}, in the memory of the closest processor socket [19]. We illustrate an example of such an architecture in Figure 4 (*right*). Our NUMA implementation requires (1) ghost-cells to exchange thread-boundary data; (2) an additional barrier to preserve the serial code semantics; and (3) that we also try to bind threads to sockets in a way that physically matches the layouts of the cores and sockets. For instance, we would map blocks {0, 1, 4, 5} in Figure 4 (*left*) to socket 0, blocks {2,3,6,7} to socket 1.

Much more extensive tuning for the CPU is possible, as discussed by others [8, 11, 2]. Our intent here is *not* to compare the CPU and GPU. Instead, we aim (a) to provide the reader with a useful baseline CPU implementation for rough comparison to a GPU, and (b) to provide our subsequent hybrid CPU/GPU implementations with a "reasonably tuned" sequential and parallel CPU components.

3.2 GPU implementations

We consider a variety of implementation techniques for NVIDIA CUDA-based GPU systems, based on the extensive experience of others [18, 1].

We treat current-generation NVIDIA GPUs using the following hardware abstraction, as proposed by others [18]. First, a GPU is a multiprocessor system with a memory hierarchy consisting of a *device memory* (the main memory on the GPU), a *shared memory*, which is a local-store shared among blocks of threads. There is also a special *texture memory* for read-only data, which has comparable latency to shared memory but fewer capacity and alignment constraints. Secondly, each multiprocessor may be viewed as either a massively multithreaded processor *or*, more accurately, as

a vector processor.

Informed by this view, we tuned our GPU implementation by considering following techniques in turn.

A baseline ("naïve") GPU implementation. The 2-D block partitioning used for the Pthreads implementation (Section 3.1) is straightforward to implement on a GPU. We divide the domain as shown in Figure 4 (left), where each subblock is assigned to a CUDA thread block, with a logical thread assigned to each unknown. This implementation mirrors the CPU-only case: we maintain two grids in the device memory for the "current" and "new" grid point values, respectively, and logically swap these grids at each iteration. The block size is a tuning parameter, which can be chosen through either manual analysis, use of CUDA diagnostic tools (*e.g.*, occupancy calculator), or experiment.

The pseudocode that executes on the CPU looks roughly as follows:

1: Copy the grid from main host memory to the GPU.
2: **for** $t \leftarrow 1 \ldots T$ **do**
3: Invoke GPU kernel for all "thread-blocks."
4: (Implicit) Synchronize host and GPU.
5: Logically swap active grid.
6: **end for**
7: Copy grid results from GPU to host memory.

Line 3 invokes execution of 1 iteration on the GPU, which corresponds to a 2-D block parallel implementation. The synchronization in line 4 occurs implicitly when the invoked kernel returns.

Though simple to write, this baseline suffers from at least two performance problems. First, it ignores the GPU memory hierarchy by not explicitly making use of

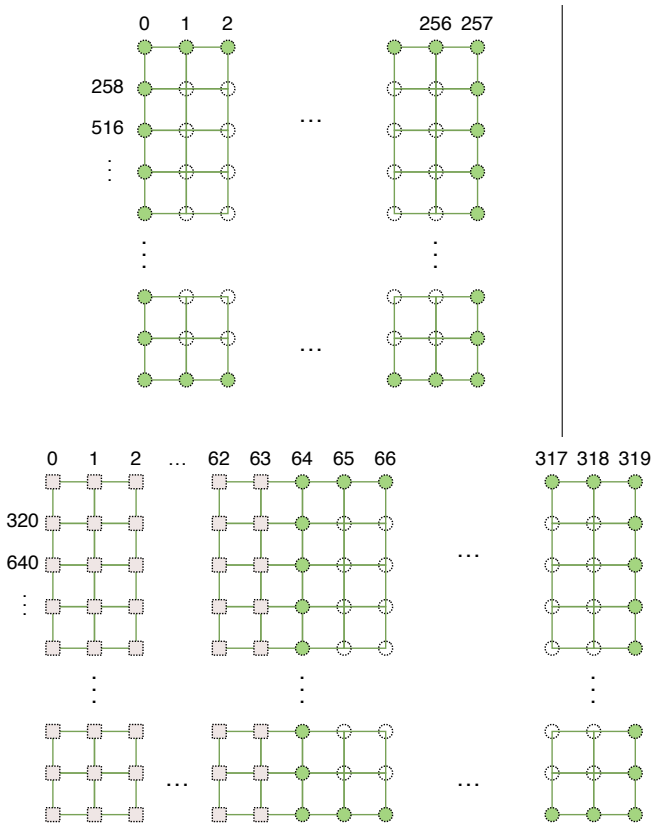


Figure 5: (*Left*) A conventional row-major layout, for $n = 256$ (258×258 grid), which could lead to costly non-coalesced memory accesses on a GPU. (*Right*) A padded row-major layout, for $n = 256$, avoids the problems of the conventional layout.

shared memory. Secondly, it will usually suffer from non-coalesced memory accesses, which are essentially unaligned vector memory operations.

Padding. We can avoid non-coalesced memory accesses by padding the grid(s) in the GPU memory to guarantee 256-byte aligned memory accesses. We illustrate padding in Figure 5 where, assuming a row-major layout, we simply store extra elements at the beginning of each row.

Shared memory, without padding. For each thread-block, we explicitly allocate a block of local-store ("shared") memory that all threads in the associated thread-block share. This block is stored in row-major order, and includes ghost cells to hold elements from neighboring blocks. The kernel computation now consists of

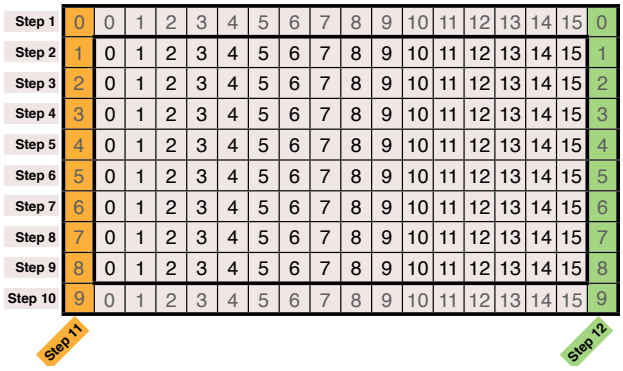


Figure 6: Thread block assignments, for a 16 thread thread-block operating on a 8×16 block of unknowns (plus fringe elements).

three phases: (A) copying the block and fringe/boundary elements from device memory to shared memory; (B) computation; and (C) writing the updated unknowns back to the grid in global memory. Compared to the baseline, this code reduces the total number of device memory fetches by $3x$ *and* we can eliminate storage of the second grid.

We must carefully assign threads to words of data, to avoid shared memory *bank conflicts*. For a thread block consisting of a particular number of threads, Figure 6 shows how we assign threads to unknown elements within the block during the reading phase (A). Each interior square represents an unknown, numbered by the thread assigned to read that element during phase (A). However, we cannot avoid bank conflicts (or even non-coalesced reads) for the left and right fringes. Thus, we expect to try to make a block as large as possible while minimizing the number of rows in each block. One may argue that we could have 2 more threads and fetch left and right fringes while doing a row access (steps 1 to 10 in Figure 6) and leave the two threads idle during computation. Such an access pattern will either reduce the occupancy of multiprocessors or cause a lot of non-coalesced, depending on the number of threads.

Texture memory. A read-only texture cache is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region

of device memory; each multiprocessor accesses the texture cache via a texture unit. The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache [1]. We consider binding the global memory to 1-D textures.

Shared memory with padding. To reduce non-coalesced reads during phase (A) above, we can again pad the grid in device memory. In addition, optimum allocation of shared memory per thread block is necessary to achieve good performance.

Unrolling. We also explicitly request unrolling of the innermost loops of the computation using the appropriate CUDA directive, and tune the unrolling depth.

A double-precision trick. Using double-precision (8-byte words) leads to 2-way bank conflicts during shared memory accesses, because the banks are arranged in a way that favors vector loads on 4-byte words. We avoid this problem by separately storing the lower and upper 4-byte words, and recombining them prior to computation using a pre-defined CUDA macro (`__hiloint2double()`) [1].

3.3 Hardware Platforms

We use the hardware evaluation platforms shown in Table 1. We consider two systems: (1) one with two 2.33 GHz dual-core Intel E6550 "Conroe" processors and two NVIDIA GPUs, a Tesla C1060 and a Quadro FX 570; and (2) another system with four quad-core AMD Opteron 8350 "Barcelona" processors and an NVIDIA Tesla C870. We use gcc 4.3.2 with the `-O4 -mtune=XXX` flags to compile our CPU implementations, and the CUDA 2.0 SDK for our GPU implementations.

Our implementation can take $f_{i,j}$ in Figure 1.1 to be either a separate grid containing arbitrary values or an inline function evaluated for any (i, j) at run-time. For our experiments, we use an inline function corresponding to a spike in the middle of the domain.

When reporting Gflop/s, we use the value of 5 flops per unknown.

Feature	NVIDIA Tesla C1060	NVIDIA Tesla C870	NVIDIA Quadro FX 570	Intel Core2Duo E6550 "Conroe"	AMD Opteron 8350 "Barcelona"
Number of multiprocessors	30	16	2	2	4
Total no. of cores	240	128	16	4	16
Peak bandwidth GB/s	102	76.8	12.8	10	21.6
Empirical streaming bandwidth (GB/s)	68.7	53.0	5.5	4.7	9.9
Double-precision?	Yes	No	No	Yes	Yes
Peak GFlop/s (Single-precision)	933	512	44	74.6[1]	256 [2]
Peak GFlop/s (Double-precision)	78	N/A	N/A	37.8	128

Table 1: Hardware platforms used in our experimental evaluation. "Empirical streaming bandwidth" measured using NVIDIA's `bandwidthTest` utility and McCalpin's STREAM Triad, as appropriate. Note that NVIDIA's bandwidth test benchmark reports "GB/s" assuming 1 GB = 1024^3 bytes, whereas we instead use the more conventional method of computing the rate via "bytes times 10^{-9} divided by time."

For reference, the baseline CPU implementations described in Section 3.1 achieve up to 4.8 Gflop/s on 16 cores of the Barcelona platform.

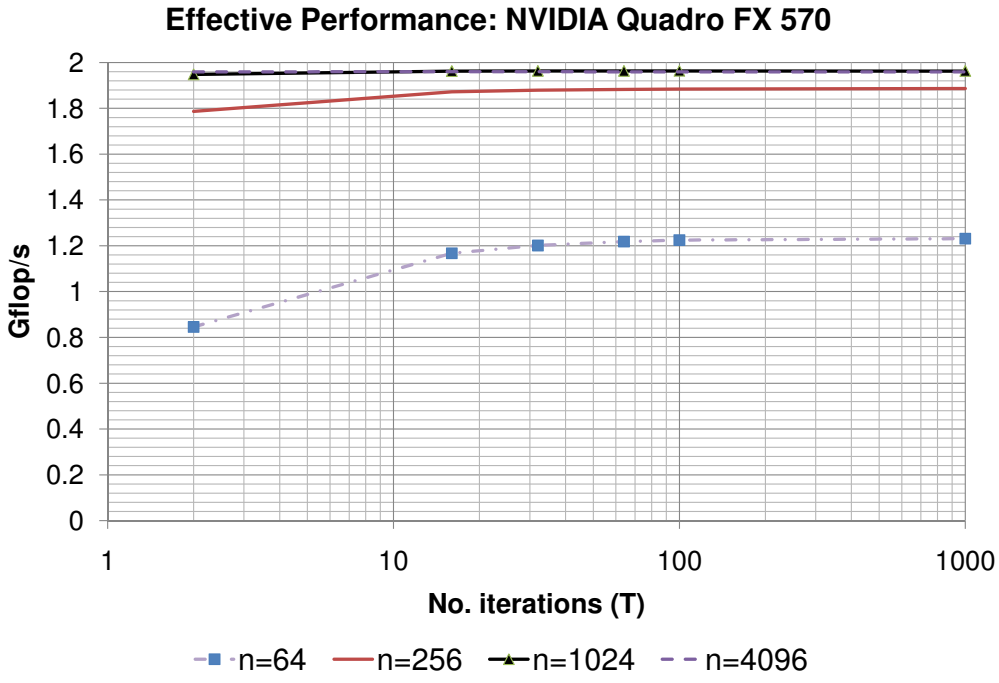
3.4 Results

We manually tuned GPU implementations on all three GPU cards, using the techniques of Section 3.2. Figure 7, Figure 8 and Figure 9 show we achieve up to 62.8% of empirical streaming bandwidth on the Quadro FX 570, up to 78.5% on the C870, and up to 98.6% on the C1060, when the grid fits within the GPU's device memory. The empirical streaming bandwidth is the bandwidth measured using the bandwidthTest benchmark in the CUDA SDK kit. These performance numbers translate into up to 1.96 Gflop/s, 22.8 Gflop/s, and 37.1 Gflop/s, respectively. However, performance is greatly diminished as the problem size falls below $n = 1024$.

From Figure 10 we see that, on the C1060, our double-precision implementations achieves 17.0 Gflop/s and an effective sustained bandwidth of 90.3% of the empirical streaming bandwidth. Compared to 98.6% bandwidth for single-precision, the difference reflects the penalty of having to pack and unpack double values.

We use a 8×8 thread block with 64×8 unknowns per thread-block to reach the best performance. These particular values minimize non-coalesced memory accesses and yield high occupancy values. It is important to get the block size right; for example, using a 16×8 block size increases the execution time by a factor of $1.7\times$ over the optimal 64×8 case on the C1060 (in single-precision).

Which of our tuning techniques had the most impact on performance? Figure 11 breaks down the final tuned GPU performance into its parts for $n = 4096$. (Performance was largely independent of the number of iterations, T, ignoring initial and final host-device copies.) Looking across generations (oldest = FX 570, newest = C1060), there are no clearly discernable trends. For example, shared memory with padding—which reduces non-coalesced accesses—is absolutely necessary on the two



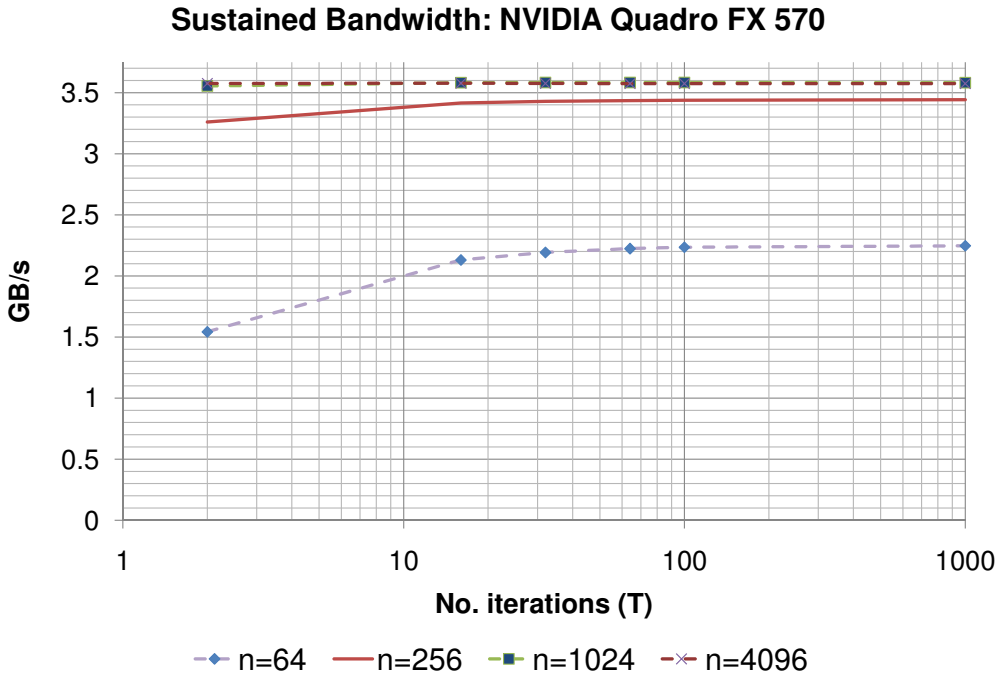


Figure 7: Sustained performance (Gflop/s) and bandwidth (GB/s) for NVIDIA Quadro FX 570, as a function of grid size (n) and number of iterations(T).

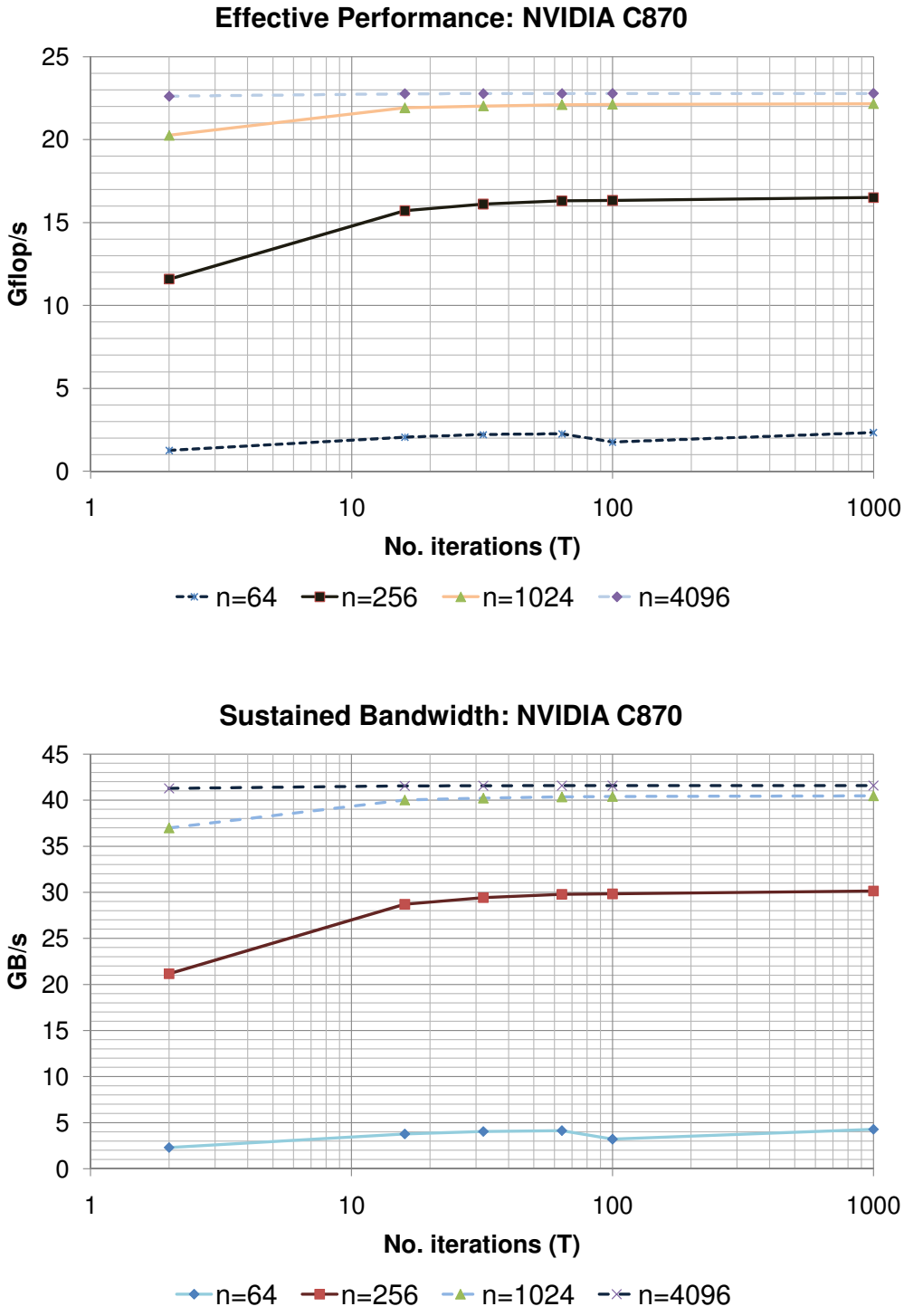


Figure 8: Sustained performance (Gflop/s) and bandwidth (GB/s) for NVIDIA C870, as a function of grid size (n) and number of iterations(T).

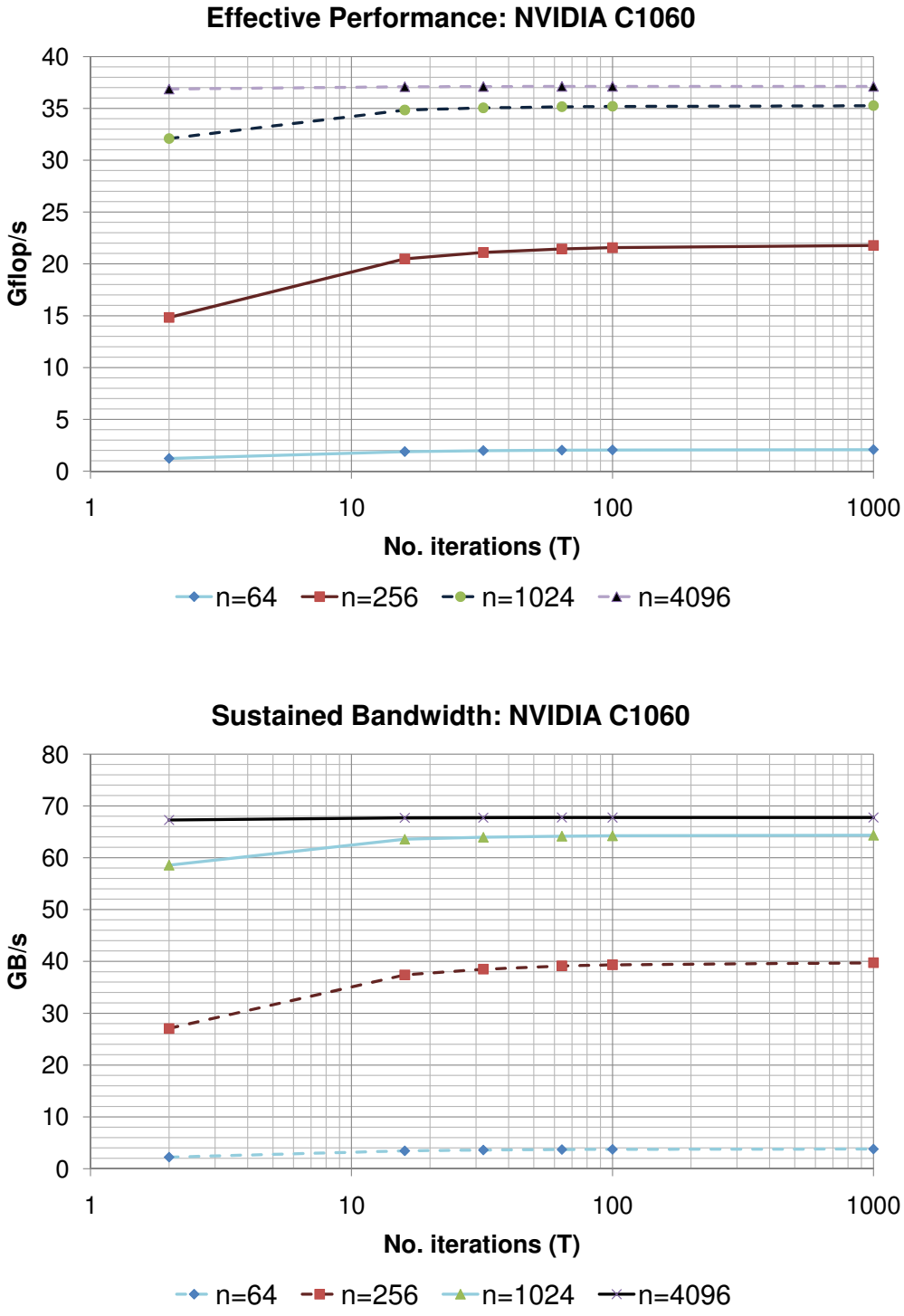


Figure 9: Sustained performance (Gflop/s) and bandwidth (GB/s) for NVIDIA C1060, as a function of grid size (n) and number of iterations(T).

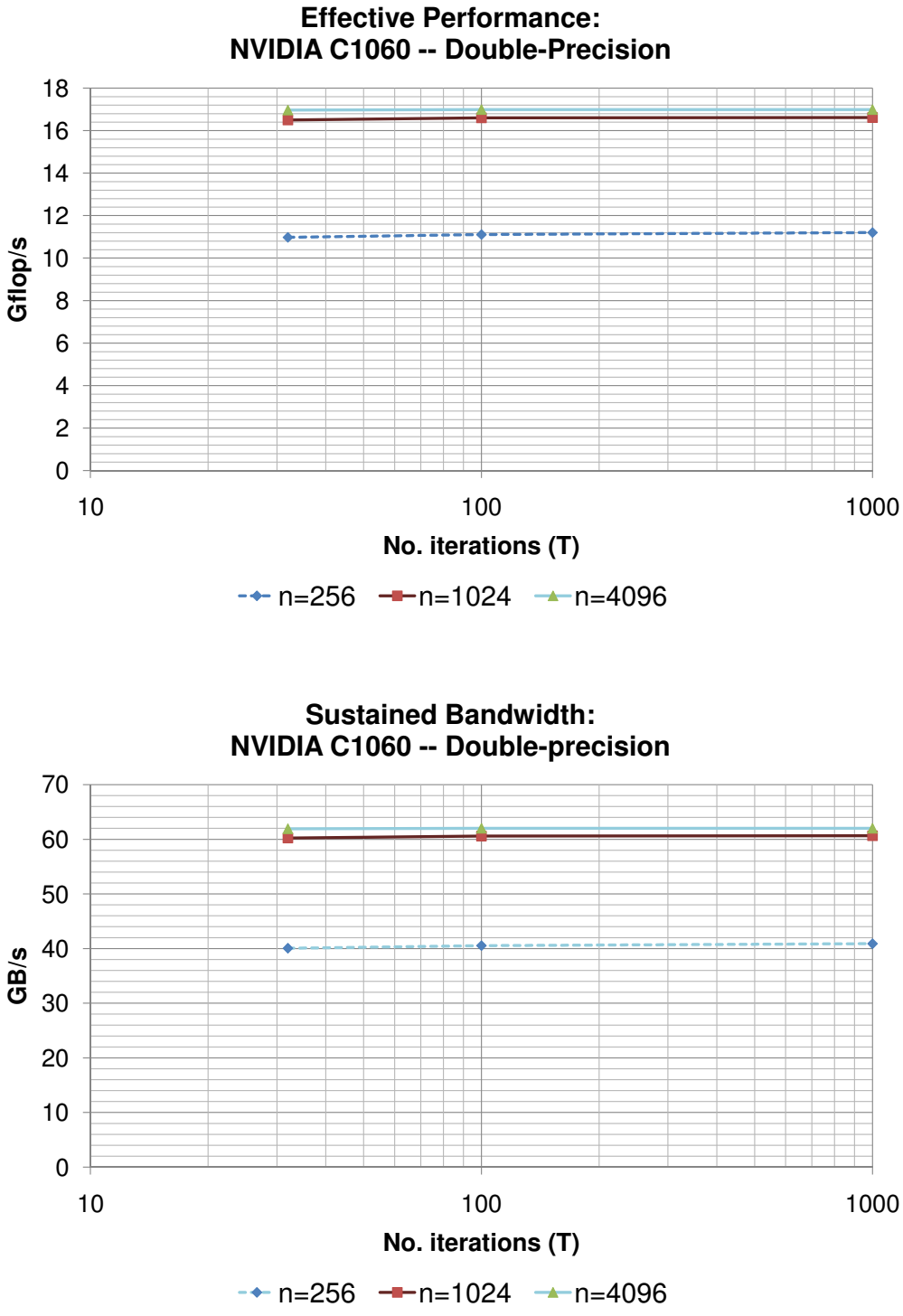


Figure 10: Sustained performance (Gflop/s) and bandwidth (GB/s) for double precision for NVIDIA C1060, as a function of grid size (n) and number of iterations(T).

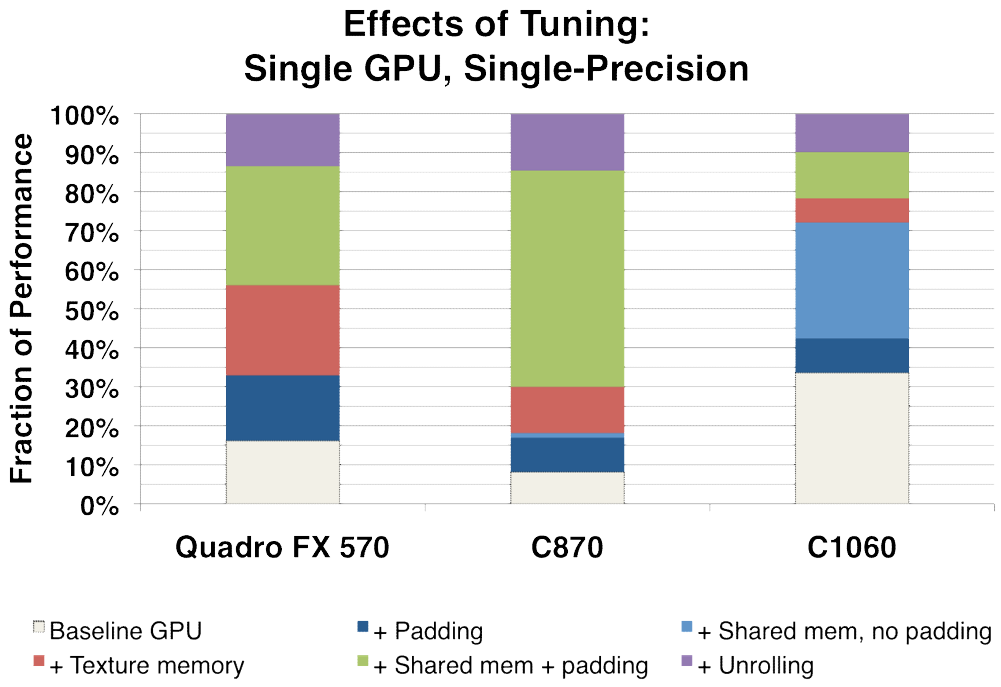


Figure 11: Cumulative impact of various tuning techniques on our implementation's final (best) performance.

older cards, but provides a relatively small improvement on the C1060. It is also worth mentioning that our experiments with prefetching data in shared memory did not yield better performance.

The preceding data ignore the *initial* and *final* data transfers between host and device, which are critical in practice. We analyze these transfers in Figure 12. When $n = 4096$ and $T = 32$ on our Barcelona+C870 platform, these transfers account for just under 44% of the total execution time, a substantial cost. (The effective host-to-device transfer rate is 1.2 GB/s.) Figure 12 (right) examines the total execution time as a function of T, to see when a GPU implementation—including the transfers—can beat a multicore CPU implementation. In this case, we need at least 5 iterations to match a tuned 16-thread CPU implementation, and 60–100 iterations to hide most of the transfer cost.

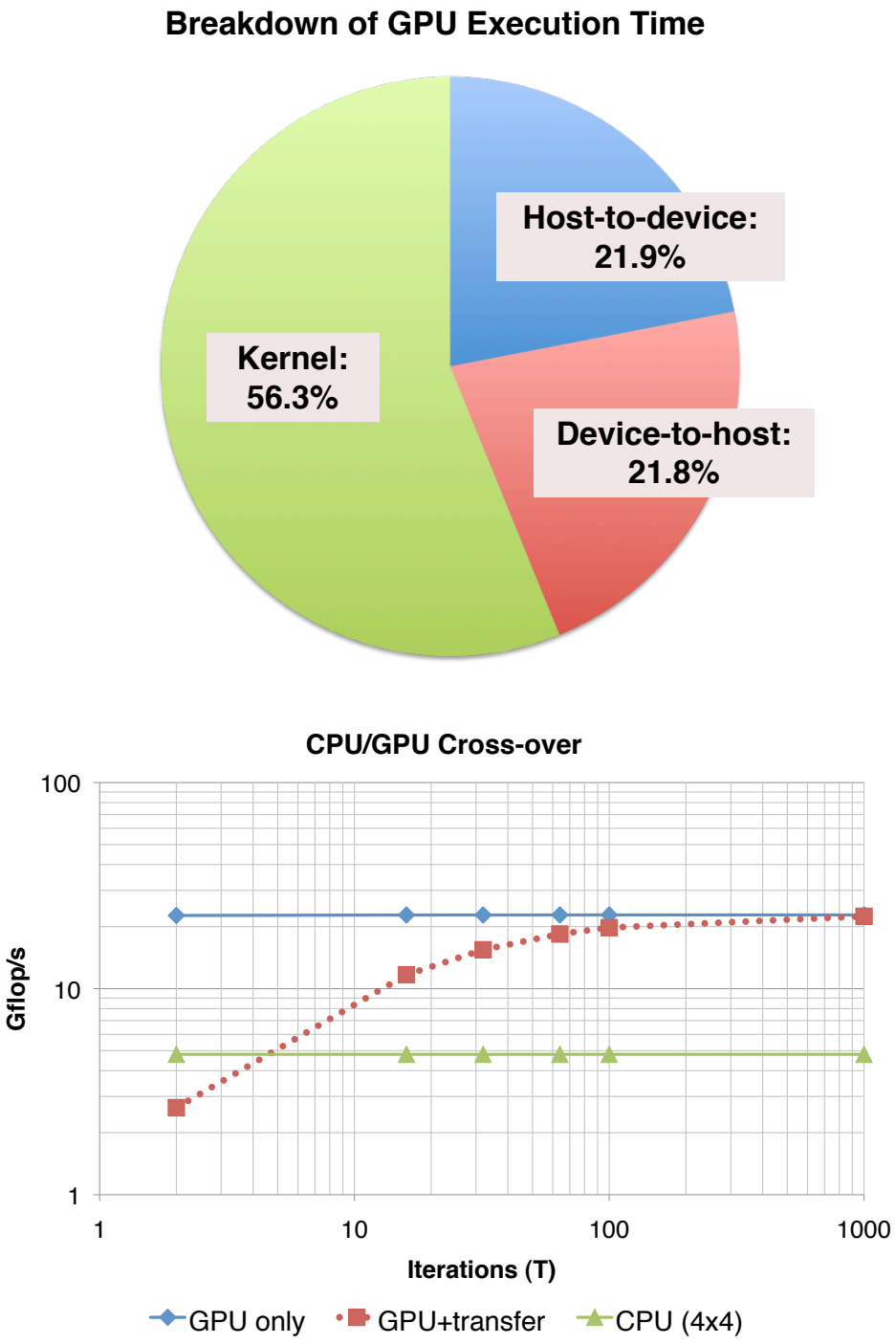


Figure 12: (*Left*) Breakdown of the total execution time using a single GPU (NVIDIA C870). Grid size is $n = 4096$, and number of iterations $T = 32$. A substantial amount of time is spent just transfering data between the host memory and GPU memory. (*Right*) Number of iterations needed for the single-GPU (NVIDIA C870) performance to exceed the baseline parallel CPU performance (Barcelona 4×4) when the initial and final grid transfers between host and device are taken into account. Grid size is $n = 4096$.

CHAPTER IV

HETEROGENOUS CPU/GPU AND HYBRID MULTI-GPU IMPLEMENTATIONS

In this chapter we consider multi-CPU variants, single- and multi-GPU variants, hybrid CPU/GPU designs.

4.1 Hybrid implementations

Hybrid CPU/GPU implementations help us avoid idle CPUs and/or GPUs. We consider a very basic strategy in which we assign a block of rows to the CPU(s), with the remaining rows assigned in blocks to available GPUs. For T Jacobi iterations on an $n \times n$ grid, we illustrate our approach in Figure 13, which implements the following.

1: // **Assign rows** $1 \ldots s$ **to the CPU(s),**

2: // **and rows** $s+1 \ldots n$ **to the GPU(s).**

3: **for** $t \leftarrow 1 \ldots T$ **do**

4: Step 1 (**GPU part**): Compute one iteration of Jacobi for the last $n - s$ rows of the grid.

5: Step 2 (**CPU part**): Simultaneously compute one iteration of Jacobi on rows $1 \ldots s$.

6: Step 3 (**Exchange data**): Transfer the boundary rows between CPU and GPU.

7: **end for**

This variant wins if Step 2 can do useful work before Step 1 finishes, given the copying overhead in Step 3. We illustrate this concept in Figure 14, which shows the

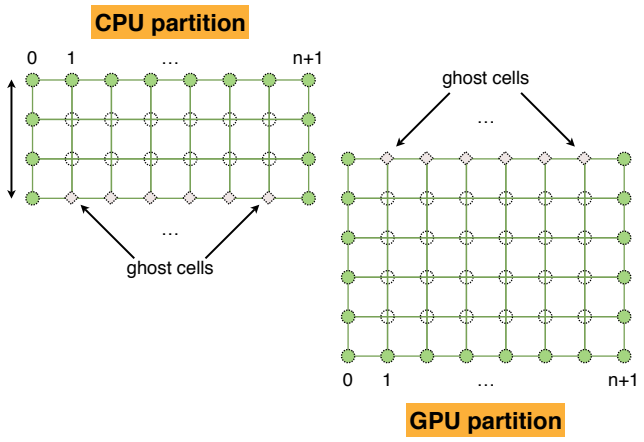


Figure 13: Block row partitioning used for the hybrid CPU/GPU implementation.

hypothetical execution times for a CPU-GPU combination in which the CPU runs at roughly $\frac{1}{3}$ the speed of the GPU. As more rows are assigned to the CPU (increasing x-axis values), the time for the CPU part (Step 2) increases while the GPU part (Step 1) decreases. Assuming perfect overlap of Steps 1 and 2, the hybrid execution time is the maximum of these two times *plus* the data exchange overhead (Step 3). To beat the presumably faster GPU-only code, we ideally want (a) sufficiently small exchange overhead, and (b) the absolute value of the slope of the CPU and GPU lines to be roughly equal. Therefore, we expect a hybrid implementation will not lead to speedups overall if there is a large gap between CPU and GPU speeds or a high transfer overhead.

For our multi-GPU and multi-CPU/multi-GPU variants, we apply the same basic principles. To use multiple GPUs in parallel, the host must dedicate one thread per GPU. Moreover, the resources for each of these GPUs must be managed inside the threads.

Figure 14 is essentially a performance model that can be used to guide multi-CPU/multi-GPU work distribution in practical settings. The component and overhead curves that vary as a function of work distribution serve as the input to the model. These curves could be non-linear or empirically statistically modeled using

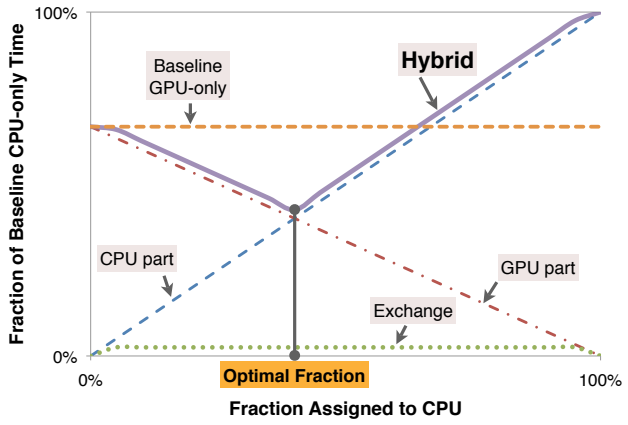


Figure 14: Illustration of expected performance of a hybrid CPU/GPU implementation.

off-line experiments, and quickly evaluated at run-time to decide on a splitting.

4.2 Results

We consider two hybrid CPU/GPU implementations in Figure 15: Conroe + Quadro FX 5703 and Barcelona + C870. We see a small 8% improvement from a hybrid implementation on the relatively slow Conroe + Quadro FX system (11% of rows assigned to the CPU), which qualitatively resembles our expectations in Figure 14. This improvement is small because the CPU is considerably slower than the GPU, as reflected in the relative slopes of the CPU-part and GPU-part lines. On the Barcelona + C870 system, we see no improvement, in large part due both to the gap in CPU/GPU processing power, as well as the data transfer overheads.

We tested our multi-GPU (no CPU) on two systems. On a FX 570 + C1060 system we observed no speedup, as is evident in Figure 16. This result is due largely to the $\approx 18\times$ gap between the fast and slow GPU. On a system with 2 C1060 cards, we were able to see a speedup of $\approx 1.8\times$

These hybrid-case findings were not surprising in light of our performance model (Figure 14). At the very least, it should be simple to instantiate this model automatically given a new hardware platform, to determine whether a hybrid implementation

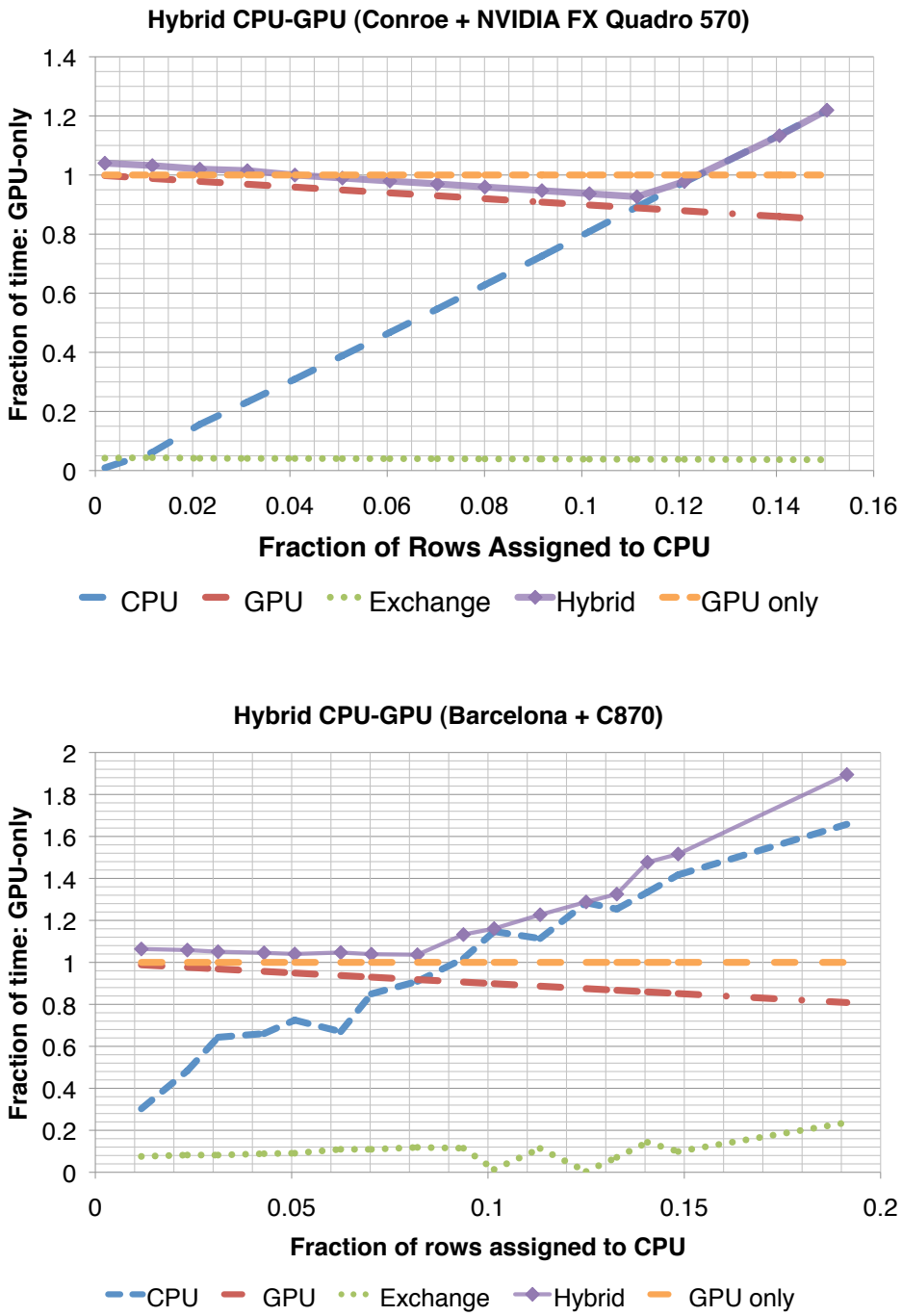


Figure 15: Measured time for hybrid multi-CPU and single-GPU implementations, normalized to GPU-only time. Compare to our hybrid performance model, illustrated in Figure 14. (*Left*) Intel single-socket dual-core Conroe + NVIDIA Quadro FX 570. At approximately 11% of rows assigned to the CPU, there is a small $\approx 8\%$ speedup over the GPU-only code. (*Right*) AMD quad-socket quad-core Barcelona (16 threads) + NVIDIA C870. The hybrid code never beats the GPU-only code due to the data exchange/synchronization.

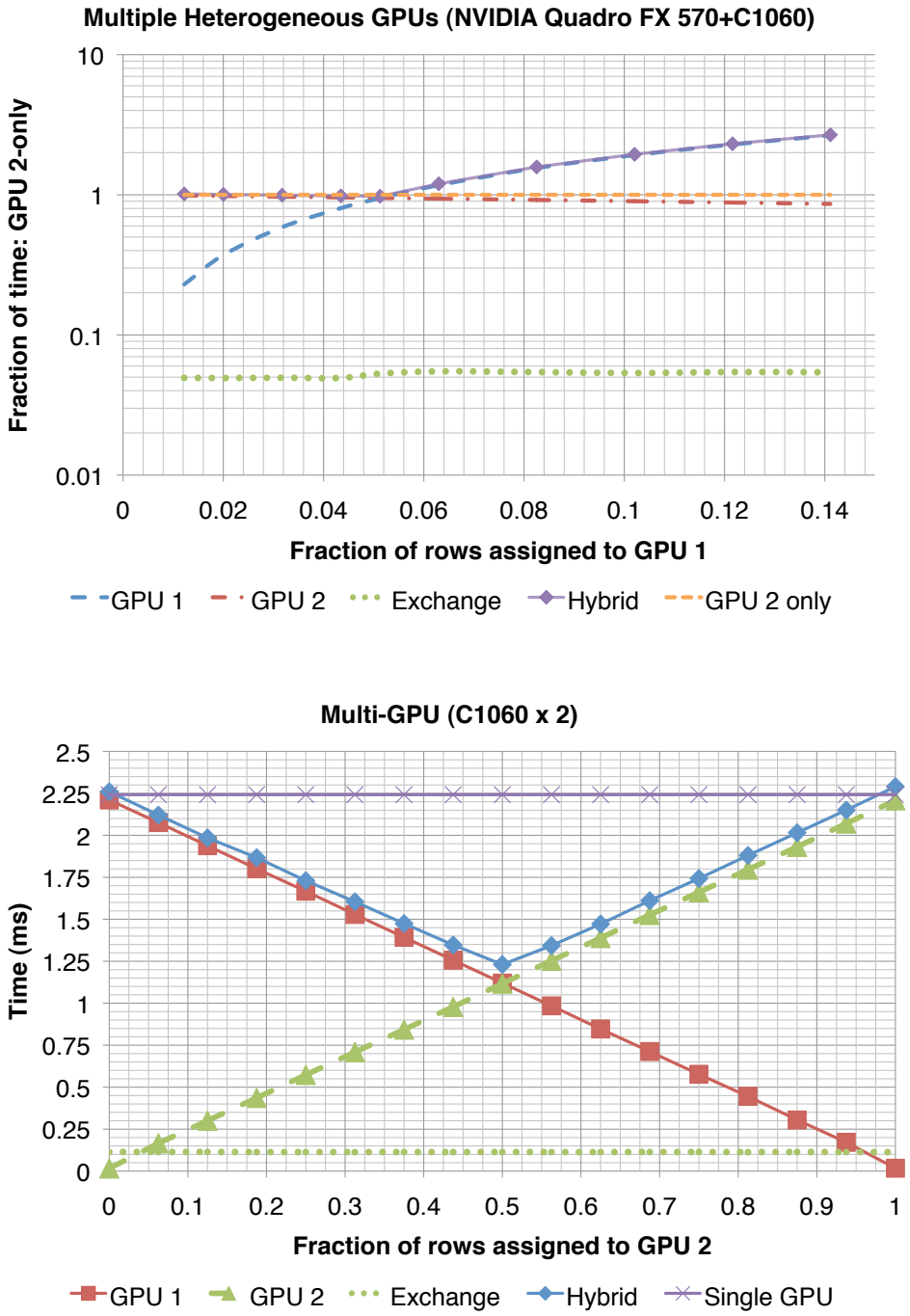


Figure 16: Measured multiple GPU performance, (*Left*) One NVIDIA Quadro FX 570 ("GPU 1") and One NVIDIA C1060 ("GPU 2"). Because of the large gap between the performance of the two GPUs ($\approx 18\times$ difference, not shown), the optimal fraction does not beat the GPU 2-only code. (Compare to Figure 14.) (*Right*) Two NVIDIA Tesla C1060 cards. At approximately 50% of rows assigned to GPU 1 there is a speedup of $\approx 1.8\times$ over the GPU-only code.

could outperform a homogeneous multi-CPU or single-GPU variant.

CHAPTER V

ASYNCHRONOUS ALGORITHMS

In Section 3.2, we described a tuned GPU-only implementation that uses padding to reduce non-coalesced device memory accesses, a proper access pattern to reduce bank conflicts, and a judicious allocation of shared memory within a thread-block to increase occupancy. We shall refer to this tuned GPU-only variant with "conventional" synchronizations as the **TunedSync** variant. In this chapter, we consider "wildly asynchronous" variations that attempt to improve performance further by *changing* the computation and synchronization. Since Jacobi's method is an iterative algorithm with many deterministic variations (*e.g.*, Gauss-Seidel [9]) that reorder updates, we expect to be able to "loosen" synchronization and still get an algorithm that converges.

5.1 Computational model

In this section we describe a simple computational model for the Jacobi's method with respect to the CUDA programming environment. Suppose the unknowns are represented as a matrix shown below.

$$\begin{pmatrix} A_1 & \dots & A_i \\ \vdots & \ddots & \vdots \\ A_j & \dots & A_m \end{pmatrix}$$

Here, A_i represent a sub-matrices of unknowns. Let us refer to these sub-matrices as components. One can think of these components as corresponding to a block of unknowns which are processed by a single thread block. Let us consider there are

m such components. Suppose there are p processing units. Let each processing unit be capable of processing d components simultaneously. The reason there is a limit on the number of components that can be processed simultaneously is because of the limited availability of shared memory and registers. This means, for a given GPU with p processing units, the GPU can process $p \times d$ components simultaneously.

There is no assurance regarding the order in which the components will be processed by the processing units. The components are always processed as a whole and there is no partial processing of components. The components may in turn be processed by multiple threads while processing. The following hypotheses must hold for a given asynchronous variant in order for it to reach convergence [10].

1. As the computation proceeds eventually one reads newer information for each of the components.

2. No component fails to be updated as time goes on.

Consider the common theoretical model for all asynchronous variants shown below:

1: // **CommonModel:**

2: Until convergence do the following

3: Execute the asynchronous kernel X on all m components using the p processing units

4: (Implicit) Sync CPU/GPU; logically swap grids.

For each variant, just replace the kernel X in the above algorithm by the appropriate kernel. We will try to infer how each asynchronous variant obeys the above hypotheses using the above model as we discuss each of them.

5.2 Review: TunedSync

First, we summarize the basic tuned GPU implementation of Section 3.2 for an $n \times n$ grid executing T Jacobi iterations as follows. Assume that we assign $R \times C$ blocks of unknowns to each CUDA thread-block, with R threads allocated per thread-block. It is straightforward to prove that the above three hypotheses hold true for this algorithm.

1: // **TunedSync:**
2: Transfer $(n+2) \times (n+2)$ grid to device memory.
3: **for** $t \leftarrow 1 \ldots T$ **do**
4: Execute the tuned GPU kernel (see below).
5: (Implicit) Sync CPU/GPU; logically swap grids.
6: **end for**

The tuned GPU kernel uses padding and thread assignment:

1: // **Tuned GPU CUDA Kernel:**
2: Declare $(R+2) \times (C+2)$ shared memory block.
3: Fetch elements from device memory to shared block, incl. fringes.
4: Synchronize threads (`sync_threads`) within the thread block.
5: Compute 1 iteration for $R \times C$ unknowns in parallel and write to device memory.

5.3 Asynchronous variations

TunedSync requires, at every iteration: (a) 1 "global" synchronization, which occurs implicitly when the GPU kernel returns; (b) one "local" synchronization (`sync_threads`); and (c) device memory reads and writes for every grid element. We seek variations

that are exactly or even only approximately equivalent to TunedSync variant and that can reduce these costs.

Async 0 and the async factor, α: All asynchronous variations we consider are parameterized by α, an *asynchronicity factor*, or just "async factor." To explain α, consider **Async 0** in Figure 17, our first asynchronous variant, which tries to replace the T global synchronizations with $\approx T/\alpha$ such synchronizations. Intuitively, we will use α as an approximate measure of the degree to which we are willing to allow threads to get "out of sync."

Async 0 differs from TunedSync in three major ways.

1. Async 0 reduces the number of device memory accesses, operating on the $R \times C$ unknowns entirely in shared memory (lines 8–20).

2. Async 0 executes lesser number of global synchronizations, replacing them with some number of thread-block-level `sync_threads` calls that depends on α.

3. The *effective number of iterations* executed by Async 1 is $T_{\text{eff}} = \frac{T}{\alpha} \cdot (\alpha + 1)$, meaning that Async 0 performs slightly more flops than TunedSync (line 22).

The thread block level barrier (14, 20) along with the read/write fringe elements statements (11, 13, 17, 19) of the algorithm ensure that hypothesis 1 is obeyed. The global synchronization ensures that hypothesis 2 is obeyed.

At this stage, it is not obvious whether Async 0 should be faster or slower than TunedSync.

Async 1. This variant further relaxes the number of synchronizations. The outer algorithm is identical to Figure 17, but we replace the thread-block GPU kernel with the one shown in Figure 18. In essence, it eliminates the 4 of the 6 local synchronizations in Async 0 (Figure 17, lines 10, 12, 16, and 18; the remaining 2 local syncs are needed to obtain convergence). However, we have also introduced an element of non-determinism in that individual threads could access or modify an

```
// Async 0:
// Threads / block is R
// Assign R × C unknowns to each thread-block
Transfer (n + 2) × (n + 2) grid to GPU.
for t ← 1 ... T_eff/α do
    Execute Async 0 GPU-Kernel.
    (Implicit) Sync CPU-GPU.
    Logically swap grids.
end for
Transfer (n + 2) × (n + 2) grid to GPU.
```

```
// Async 0 GPU-Kernel:
// Executes on all thread-blocks
Declare two (R + 2) × (C + 2) shared memory grid blocks, B_1 and B_2.
Fetch (R + 2) × (C + 2) elements from device memory into B_1.
Copy just the fringes to B_2.
sync_threads: Sync thread-block.
// Inner α loop is "unrolled" by 2
for v ← 1 ... α/2 do
    Compute 1 iteration in B_1, writing to B_2.
    sync_threads
    Write penultimate fringe from B_2 to device memory.
    sync_threads
    Fetch fringe elements from device memory to B_2.
    sync_threads
    Compute 1 iteration in B_2, writing to B_1.
    sync_threads
    Write penultimate fringe from B_1 to device memory.
    sync_threads
    Fetch fringe elements from device memory to B_1.
    sync_threads
end for
Compute one Jacobi step with elements in B_1.
Write results back the results to the device memory.
```

Figure 17: Algorithm: Async 0. This algorithm is the basic skeleton in which we consider removing synchronizations and/or device memory accesses to create other "fast-and-loose" variants. Here, the "penultimate fringe" is the boundary of unknowns (outermost ring of unknowns bordering the ghost cells) that neighboring thread-blocks will need.

1: // **Async 1 GPU-Kernel:**
2: // **Executes on all thread-blocks**
3: Declare two $(R+2) \times (C+2)$ shared memory grid blocks, B_1 and B_2.
4: Fetch $(R+2) \times (C+2)$ elements from device memory into B_1.
5: Copy just the fringes to B_2.
6: `sync_threads`: Sync thread-block.
7: **for** $v \leftarrow 1 \ldots \alpha/2$ **do**
8: Compute 1 iteration in B_1, writing to B_2.
9: Write penultimate fringe from B_2 to device memory.
10: Fetch fringe elements from device memory to B_2.
11: `sync_threads`
12: Compute 1 iteration in B_2, writing to B_1.
13: Write penultimate fringe from B_1 to device memory.
14: Fetch fringe elements from device memory to B_1.
15: `sync_threads`
16: **end for**
17: Compute one Jacobi step with elements in B_1.
18: Write results back the results to the device memory.

Figure 18: Algorithm: Async 1. (GPU-Kernel only) This variant eliminates 4 of the 6 local syncs in Figure 17.

unknown mix of old and new values through the asynchronous fringe element accesses. Since we have eliminated a number of synchronizations, we hope that we can trade additional flops via $T_{\text{eff}} > T$ and for less time while still achieving the same level of accuracy as TunedSync with T iterations.

The thread block level barrier (11, 15) along with the read/write fringe elements statements (9, 10, 13, 14) of the algorithm ensure that hypothesis 1 is obeyed. The global synchronization ensures that hypothesis 2 is obeyed.

Async 2. This variant, shown in Figure 19, has the same number of synchronizations as Async 1 but eliminates fringe reads and writes to reduce the number of device memory accesses. We expect to need a larger T_{eff} than Async 1 but hope it is offset by the speed improvements due to fewer device memory accesses.

The thread block level barrier (9, 11) ensures that hypothesis 1 is obeyed. The global synchronization ensures that hypothesis 2 is obeyed. Suppose we try to move all the global synchronization barriers inside the kernel and make them thread-block

1: // **Async 2 GPU-Kernel:**
2: // **Executes on all thread-blocks**
3: Declare two $(R+2) \times (C+2)$ shared memory grid blocks, B_1 and B_2.
4: Fetch $(R+2) \times (C+2)$ elements from device memory into B_1.
5: Copy just the fringes to B_2.
6: `sync_threads`: Sync thread-block.
7: **for** $v \leftarrow 1 \ldots \alpha/2$ **do**
8: Compute 1 iteration in B_1, writing to B_2.
9: `sync_threads`
10: Compute 1 iteration in B_2, writing to B_1.
11: `sync_threads`
12: **end for**
13: Compute one Jacobi step with elements in B_1.
14: Write results back the results to the device memory.

Figure 19: Algorithm: Async 2. (GPU-Kernel only) This variant eliminates the fringe writes and reads in lines 9, 10, 13, and 14 of Figure 18.

level barriers, we might never reach convergence. This is because we might violate hypothesis 2. We will be updating only a part of the components (which can fit the GPU) as time goes on and the rest of the components will remain unupdated.

Async 3. Our final variant maintains just a single shared memory grid block, rather than two, as shown in Figure 20. This variant is the "most wild" in that we eliminate all local synchronization. This method is similar in spirit to Gauss-Seidel iteration—grid points can use the latest updated values—except that there is no deterministic sequential ordering of updates.

The read/write fringe elements statements (8, 9,) of the algorithm ensure that hypothesis 1 is obeyed. The global synchronization ensures that hypothesis 2 is obeyed.

5.4 Results

We compare Async 1, 2, and 3 implementations of Section 5 on a $n = 4096$ grid for a single-GPU C1060. The results appear in Figure 21. In all cases, we run the Async algorithms for as large a value of T_{eff} as is necessary to achieve the same accuracy as

1: **// Async 3 GPU-Kernel:**
2: **// Executes on all thread-blocks**
3: Declare *one* $(R+2) \times (C+2)$ shared memory grid block, B.
4: Fetch $(R+2) \times (C+2)$ elements from device memory into B.
5: `sync_threads`: Sync thread-block.
6: **for** $v \leftarrow 1 \ldots \alpha$ **do**
7: Compute 1 iteration in B, writing to B.
8: Write penultimate fringe from B to device memory.
9: Fetch fringe elements from device memory to B.
10: **end for**
11: Compute one Jacobi step with elements in B.
12: Write results back the results to the device memory.

Figure 20: Algorithm: Async 3. This "most wild" variant replaces the two shared memory grid blocks with 1, and eliminates all local synchronization.

TunedSync with $T = 1000$ iterations. We measure accuracy as the maximum absolute value of the relative error computed against the solution after many (4096) iterations. We evaluate two aspects of these implementations: the bottom-line speedup relative to TunedSync and the relative increase in the effective iteration count, T_{eff}. Because the algorithms are non-deterministic, we include error bars, though in all cases they are too small to see clearly.

Async 1 is never faster than TunedSync. Though there are fewer global syncs, there are several local syncs and device memory accesses for fringe elements. As shown in Figure 21 (right), Async 1 also always has $T_{\text{eff}} > T$.

Async 2, by contrast, exceeds the performance of TunedSync by up to 1.3× for async factors $\alpha \leq 15$. This best speedup, which occurs for $\alpha = 6$ (Figure 21 (left)), exists despite the fact that Async 2 is performing 1.7× as many flops (Figure 21 (right)).

The best speedups are produced by the most aggressive algorithm, Async 3. Its performance, which is up to 2.5× faster than TunedSync, is due to reduced synchronization, reduced device memory accesses, and accelerated convergence.

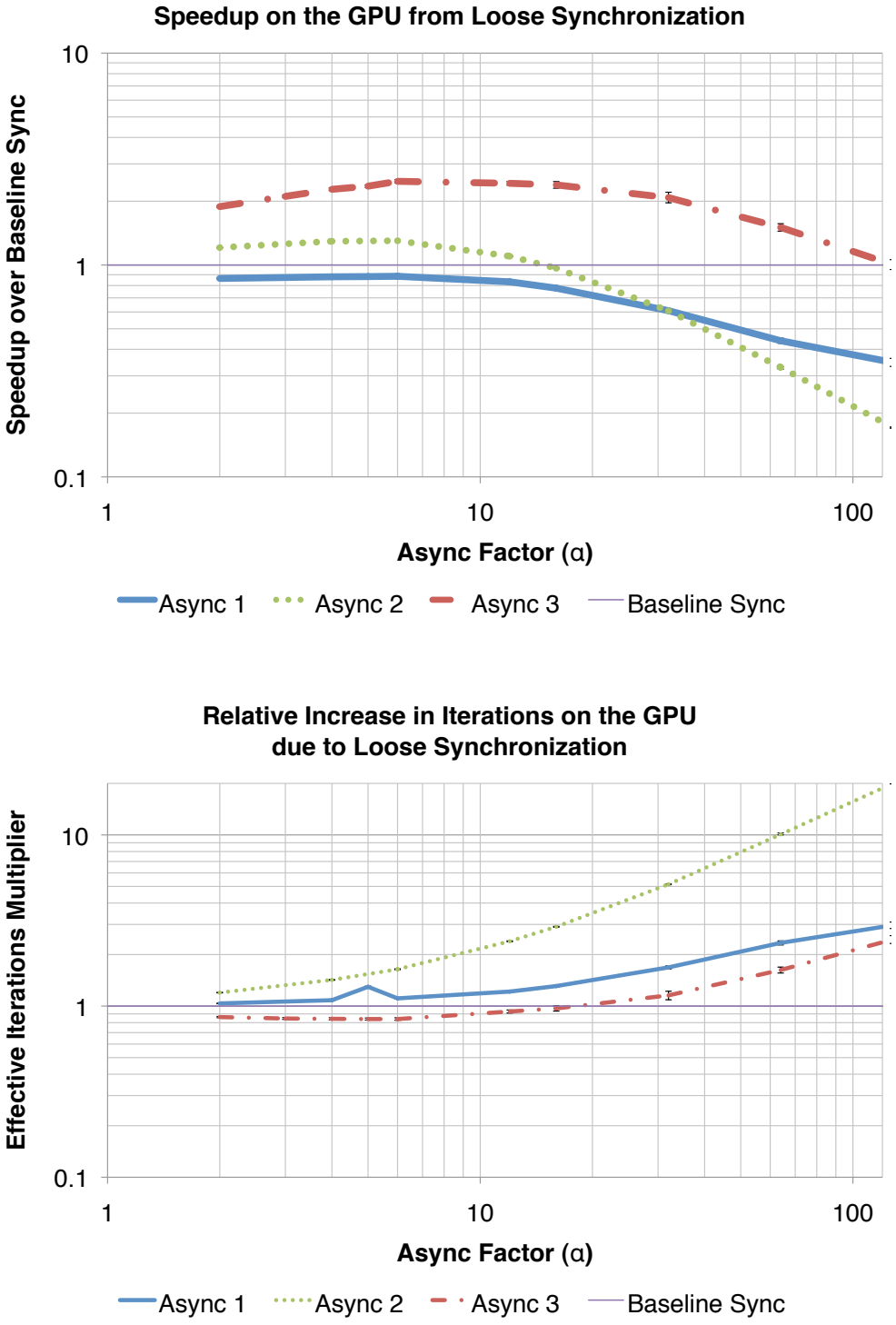


Figure 21: Comparison of asynchronous implementations on the NVIDIA C1060, for $n = 4096$ and $T = 1000$. (*Left*) Speedup relative to the synchronized baseline. (*Right*) Relative increase in effective iterations required to reach the same level of accuracy as the tuned synchronized GPU baseline (synchronized baseline = 1).

CHAPTER VI

CODE GENERATOR

In this chapter we describe a code generator for "Jacobi-like" stencils that was built. The code generator is capable of generating code automatically for given parameters of a stencil. It generates both the host code and the GPU code for the stencil.

6.1 Design

The design of the code generator is fairly straightforward. It uses a config file to control the code being generated. The code-generator might internally run multiple passes to generate the final code. It first generates the code in a macro-level language which is later converted to C-code. It exposes a config file to the user which can be used to specify the parameters for the code that should be generated. The following are the parameters that the code generator can handle:

Neighbours to be used for computation: The code generator is capable of generating code for any arbitrary stencil. The problem in Figure 1.1 shows a 2-D stencil that updates the unknowns using the values of four first-level neighbours. For stencils, in general the update can come from any number of neighbours and can be from any level. The code-generator can generate code automatically for any given configuration of a stencil. This is represented in the config file using relative co-ordinates.

Single/Double precision: The code generator can generate code for both single precision and double precision values.

GPU only/ CPU-GPU / Multi-GPU code: Specifies whether GPU only/ CPU-GPU / Multi-GPU code code has to be generated.

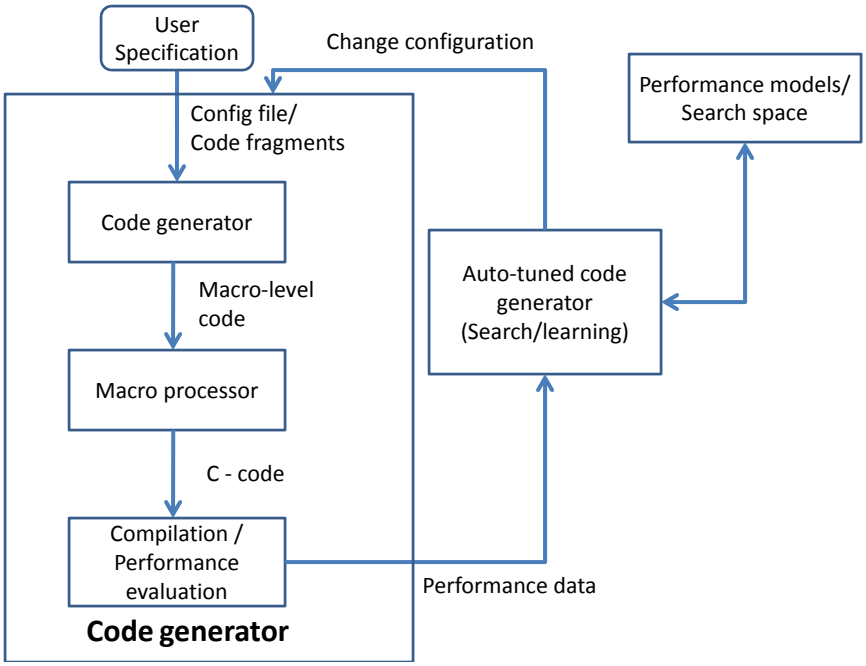


Figure 22: Architecture of the code-generator and the proposed auto-tuned code generator

Partition factor: The partitioning of rows between different devices for CPU-GPU and the Multi-GPU implementations.

Number of threads per block: This indicates the number of default threads per thread block. This can be changed in run-time while running the generated code.

Number of unknows per block: This indicates the number of unknowns that will be computed by one CUDA thread block per thread. This determines the shared memory required per CUDA thread block.

Spike factor: This is the value of h in the equation in Figure 1.1.

Analytical function $f(x)$: The analytical function used for the update as shown in Figure 1.1

Weights for the neighbours: While updating an unknown, the neighbours may

be weighed.

Target directory: This indicates the target directory in which the code needs to be generated.

The code generator initially generates a macro-like language. A macro-processor must do a second pass over the code to generate the actual C- code. Figure 22 shows the architecture of the code generator. We have built a code-generator in this work which can be extended to build an auto-tuned code generator as shown in the figure.

6.2 Auto-tuned code generator

Given the code generator in the previous section, we can develop an auto-tuned code generator for a given platform and for a given stencil. Although an auto-tuned generator was not developed in this work, given the design of the code-generator above, one can design an auto-tuned code generator using that. The auto-tuned code generator will have access to the search space of different parameters and performance models for different parameters which it uses to generate optimized code. The auto-tuned code generator changes the configuration for the code-generator in the config file, generates the code, and then the performance of the code is evaluated by compiling and executing the code. This performance feedback along with performance models will help the auto-tuned code generator change the configuration appropriately.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Though several studies exist on stencil kernels for GPU systems, our examination of heterogeneous designs, simple performance models for these designs, and exploration of fast-and-loose asynchronous algorithms extend our collective understanding in the area. In particular, two aspects of our work seem especially relevant to future systems, which will feature many heterogeneous cores.

First, we consider real costs, such as host-to-device transfer time, that have sometimes been omitted in prior work. The corresponding performance model for hybrid designs, which we presented simply to explain why our hybrid implementations did not yield significant speedups, is very simple but informative, and should be relevant to future implementations and systems.

Secondly, our evaluation of the classical idea of chaotic relaxation, though highly speculative, would seem to be a fruitful future direction in the regime of high sychronization and communication costs. The mathematical survey of Frommer and Szyld [10] summarizes known convergence conditions, and references demonstrated applications in linear and non-linear solvers and optimization, inverse problems in geophysics, and power network analysis, among others. Extending these ideas to other domains will of course require careful analysis, but we believe our results suggest that now is an appropriate time to take a fresh look into this area of research.

Finally, the simple code generator we built serves as a motivation and good starting point to build an auto-tuned code generator as described in Section 6.2. Also, the idea of automatically generating asynchronous variants for a given algorithm, evaluating their performance, and finding the "best" asynchronous variant appears to be a good

direction to move forward.

REFERENCES

[1] "NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide, Version 2.0." http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, June 2008.

[2] Amorim, R., Haase, G., Liebmann, M., and dos Santos, R. W., "Comparing CUDA and OpenGL implementations for a jacobi iteration," Tech. Rep. SFB-Report No. 2008-025, Universität Graz, Graz, Austria, December 2008.

[3] Baudet, G. M., "Asynchronous iterative methods for multiprocessors," *J. ACM*, vol. 25, pp. 226–244, April 1978.

[4] Bolz, J., Farmer, I., Grinspun, E., and Schröder, P., "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," in *Proc. ACM SIGGRAPH*, (San Diego, CA, USA), 2003. http://www.multires.caltech.edu/pubs/GPUSim.pdf.

[5] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J., "A class of parallel tiled linear algebra algorithms for multicore architectures.," Tech. Rep. UT-CS-07-600 (LAPACK Working Note 191), Innovative Computing Laboratory, University of Tennessee Knoxville, September 2007. http://www.netlib.org/lapack/lawnspdf/lawn191.pdf.

[6] Chan, E., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R., "SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," in *Proc. ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*, (San Diego, CA, USA), pp. 116–125, June 2007.

[7] Chazan, D. and Miranker, W., "Chaotic relaxation," *Linear Algebra and Its Applications*, vol. 2, pp. 199–222, April 1969.

[8] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D. A., Shalf, J., and Yelick, K. A., "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, (Austin, TX, USA), November 2008.

[9] Demmel, J. W., *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: SIAM, 1997.

[10] Frommer, A. and Szyld, D. B., "On asynchronous iterations," *J. Comp. Appl. Math.*, vol. 123, pp. 201–216, November 2000.

[11] Giles, M., "Jacobi iteration for a Laplace discretization on a 3D structured grid." http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/laplace3d.pdf, April 2008.

[12] Kale, K. G. and Lew, A. J., "Parallel asynchronous variational integrators," *Int. J. Numer. Meth. Engng.*, vol. 70, pp. 291–321, 2007.

[13] Karimabadi, H., Driscoll, J., Omelchenko, Y. A., and Omidi, N., "A new asychronous methodology for modeling of continuous systems: Breaking the curse of the Courant condition," *J. Comp. Phys.*, vol. 205, pp. 755–775, 2005.

[14] Lew, A., Marsden, J., Ortiz, M., and West, M., "Asychronous variational integrators," *Arch. Rational Mech. Anal.*, 2003.

[15] Nutaro, J. J., *Parallel discrete event simulation with application to continuous systems*. PhD thesis, University of Arizona, 2003.

[16] Strikwerda, J. C., "A probabilistic analysis of asynchronous iteration," *Linear Algebra and Its Applications*, vol. 349, pp. 125–154, January 2002.

[17] Venkatasubramanian, S. and Vuduc, R., "Tuned and wildly asynchronous stencil kernels for heterogeneous cpu/gpu systems," in *Proc. ACM International Conference on Supercomputing*, (New York, USA), June 2009.

[18] Volkov, V. and Demmel, J. W., "Benchmarking GPUs to tune dense linear algebra," in *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, (Austin, TX, USA), November 2008.

[19] Williams, S., Oliker, L., Vuduc, R., Shalf, J., and Yelick, K. A., "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, (Reno, NV, USA), November 2007.